A Framework for Dynamic Energy Efficiency and Temperature Management*

Michael Huang†, **Jose Renau**†, **Seung-Moon Yoo**‡, and **Josep Torrellas**†
Department of Computer Science†
Department of Electrical and Computer Engineering‡
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

ABSTRACT

While technology is delivering increasingly sophisticated and powerful chip designs, it is also imposing alarmingly high energy requirements on the chips. One way to address this problem is to manage the energy dynamically. Unfortunately, current dynamic schemes for energy management are relatively limited. In addition, they manage energy either for energy efficiency or for temperature control, but not for both simultaneously.

In this paper, we design and evaluate for the first time an energy-management framework that tackles both energy efficiency and temperature control in a unified manner. We call this general approach Dynamic Energy Efficiency and Temperature Management (DEETM). Our framework combines many energy-management techniques and can activate them individually or in groups in a fine-grained manner according to a given policy. The goal of the framework is two-fold: maximize energy savings without extending application execution time beyond a given tolerable limit, and guarantee that the temperature remains below a given limit while minimizing any resulting slowdown. The framework successfully meets these goals. For example, it delivers a 40% energy reduction with only a 10% application slowdown.

1 INTRODUCTION

Continuous technical advances are fueling the trend toward more sophisticated and powerful chip designs. Such designs, including high-end microprocessors, chip multiprocessors, systems on a chip, and other advanced embedded systems are quickly increasing their functionality and clock rates. Unfortunately, they are also increasing their energy consumption requirements alarmingly.

One way to address this problem is to manage the energy consumed in the chips. There are two main aims of energy management: to ensure that the energy is used efficiently and to guarantee that power consumption is never so high that the chip reaches dangerous temperature levels.

Efficient energy use is desirable in all systems. However, it is critical in portable devices, where battery energy is limited. It is also an important way to reduce cost in systems that have periods of idle time, also called slack [25]. Slack appears not only in interactive and real-time systems; it also occurs in general-purpose environments like web servers or routers with high-end processors where the performance is often bottlenecked by the network.

Likewise, curbing high power consumption to limit high temperatures is useful in all systems. It enables lower-cost packaging and cooling systems for the chips. It also makes the chip more reliable. Finally, it may enable a more aggressive design or a higher clock speed.

To address these two issues, namely energy efficiency and temperature control, many low-power architectural techniques have been proposed and implemented. For example, they include putting the system in sleep mode [28]; scaling the voltage and/or frequency [11, 13, 25]; switching contexts to a job that consumes less power [27]; reconfiguring hardware structures [1]; gating pipeline signals, for example to control speculation [5, 23]; throttling the instruction cache [28]; clock optimizations, including multiple clocks and clock gating [10]; better signal encoding [10]; low power memory design techniques [15] like bank partitioning or divided word line; low power cache design techniques like cache block buffering [33], sub-banking [9, 30], or filter caches [20]; and TLB optimizations [17].

While most of these techniques are likely to be useful for the upcoming, energy-consuming chips, we feel that their effectiveness can be enhanced. To start with, while some of these techniques have been used adaptively [1, 8, 5, 23, 25, 27], many others have been designed to be always active. In reality, for many of the latter, it would be advantageous to turn them on and off dynamically, based on the requirements of the application and the environmental conditions. They could enable useful energy-performance tradeoffs.

In addition, most of these techniques were proposed to work independently of each other. If we combined many of them in a single framework that can activate and deactivate them individually or in groups according to a given policy, the resulting system could be both more powerful and more flexible.

Finally, proposed dynamic approaches have targeted either energy efficiency [1, 8, 23, 25] or temperature control [5, 27] but not both simultaneously. If a multi-technique framework can combine support for both aspects, it can become a fairly complete approach to energy management.

The general approach of dynamically managing energy for both energy efficiency and temperature control we call *Dynamic Energy Efficiency and Temperature Management (DEETM)*. The contribution of this paper is the design and evaluation for the first time of one such DEETM framework. Our framework supports a combination of energy-management techniques. It is implemented with a combination of software and hardware for fine-grained energy management. The framework has two goals: (i) maximize the

*This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, MIP-9619351, and CCR-9970488, DARPA Contract DABT63-95-C-0097, and gifts from IBM and Intel.

savings of energy in the chip without extending the execution time of the application beyond a given tolerable limit, and (ii) guarantee that the temperature of the chip remains below a given limit while minimizing any resulting slowdown. In our evaluation, we show that the framework satisfies these goals. For example, it delivers a 40% energy reduction with only a 10% application slowdown.

This paper is organized as follows: Section 2 presents the design and implementation of our framework for DEETM; Section 3 discusses how we evaluate it; Section 4 evaluates the framework; and Section 5 presents related work.

2 A FRAMEWORK FOR DEETM

In this section, we describe our framework for DEETM: its main ideas (Section 2.1), the algorithm used (Section 2.2), the software interface (Section 2.3), some related issues (Section 2.4), and the techniques included in the framework (Section 2.5).

2.1 Main Ideas

Advanced chips can benefit from a dynamic framework that manages energy in a fine-grained manner to accomplish two goals. The first one is *temperature control*: guaranteeing that the temperature of the chip remains below a given limit while minimizing any slowdown. The second goal is *energy efficiency control*: maximizing the savings of energy in the chip without extending the execution of the application beyond a tolerable limit.

For the framework to be versatile, it should include multiple techniques for energy management. Different techniques may target the energy consumption in different components of the chip, for example processor cores, I-caches, D-caches, or DRAM arrays. They may, instead, target the same component but do so with a different energy-performance tradeoff. In such an environment, the framework can dynamically activate the techniques individually, concurrently, gradually with a priority order, or even in a mutually exclusive manner.

As initial support for the framework, we assume that the chip contains a distributed thermal sensor along the lines of the PowerPC [28] and a counter with the number of instructions executed. In addition, it contains two registers, *MaxTemp* and *MaxSlowdn*, which are set in software with the maximum temperature allowed and the maximum job slowdown that can be tolerated, respectively.

2.2 Algorithm Description

Our framework includes two algorithms: a temperature-limiting one called *Thermal* and an energy-saving one called *Slack*. They try to satisfy the first and second goals discussed above, respectively. These algorithms control the activation of a set of energy-management techniques.

At any given time, the set of techniques that are active is called the *Current Set*. These techniques may have been selected by the Thermal or by the Slack algorithm. The set of techniques that are selected by the Thermal algorithm is called the *Thermal Set*.

The two algorithms work as follows. When the Thermal algorithm runs, it compares the current temperature to the temperature limit. Depending on the result, it may add or subtract one technique to or from the Thermal Set. When the Slack algorithm runs, it first deactivates the Current Set to measure the baseline IPC value of the application. Then, it activates the Thermal Set and possibly additional techniques until the new IPC shows that the tolerable slack is used up.

To adapt to changing conditions, these algorithms run periodically. The period between runs we call *Macrocycle*. Since the two algorithms do not need to have the same period, we define a thermal macrocycle and a slack macrocycle (Figure 1-(a)).

The thermal macrocycle should be set roughly to the time taken by the thermal sensor to detect a change in temperature after a technique is activated. Since heat transfer occurs at the ms level [31], the thermal macrocycle has to be of the order of a few ms, possibly 1-15 ms. If the macrocycle is too short, the Thermal algorithm will overreact, since there is not enough time to feel the effect of any newly activated technique. However, if it is too long, we risk damaging the chip with a temperature that is over the limit for too long. The appropriate length of the macrocycle is different in each system. It depends on the heat dissipation characteristics of the chip and the sophistication of the distributed thermal sensor.

Selecting the slack macrocycle is not as delicate. However, since the Slack algorithm decides what fraction of the time to activate each technique for, based solely on the IPC at the beginning of the macrocycle, we need to pay attention to two issues. First, the macrocycle should be short enough not to miss significant changes in application behavior. Otherwise, the resulting slowdown may be very different than initially expected. In practice, a macrocycle of the order of a few ms, possibly 1-15 ms, is appropriate.

The second issue is that slack macrocycles should all have the same duration and not be cut off short. The reason is that, when the Slack algorithm runs, its calculations use the expected duration of the macrocycle to decide the length of time to activate each technique for. Cutting the macrocycle short makes such calculations inaccurate. We will see later how we address this issue.

In the following, we describe the two algorithms in detail. Note that both algorithms want to deliver large energy reductions without excessive slowdowns. Consequently, they prefer techniques that minimize the product of the energy consumed by the application times the execution time (*energy-delay product* [10]). As a result, both algorithms pick the techniques to activate in the same order. Such order follows a ranking set up by the OS or application based on the expected energy-delay product impact of each technique.

Thermal Algorithm

The Thermal algorithm is typically implemented as an interrupt handler in the OS. Alternatively, it could be implemented in hardware. The algorithm is shown in Figure 1-(b). If the thermal sensor indicates a temperature higher than *MaxTemp*, the next highest-priority technique not yet in the Thermal Set is added to it. Otherwise, if it indicates a temperature lower than a low-threshold value *MinTemp*, the lowest-priority technique in the Thermal Set is removed.

If we have added a new technique to the Thermal Set, before leaving the algorithm, we set the Current Set to the maximum of Current and Thermal Sets. This is done to ensure that the new technique is immediately active. If a technique was removed from the Thermal Set, however, it cannot be removed from the Current Set until the Slack algorithm runs.

MinTemp is set to minimize instability. A sophisticated design can keep a different *MinTemp* for each of the techniques. To choose the appropriate *MinTemp* for a given technique, we can use past profiles to estimate the temperature reduction that the technique delivers under usual conditions. Then, we set *MinTemp* to slightly less than *MaxTemp* minus the average value of such a temperature reduction. With this approach,

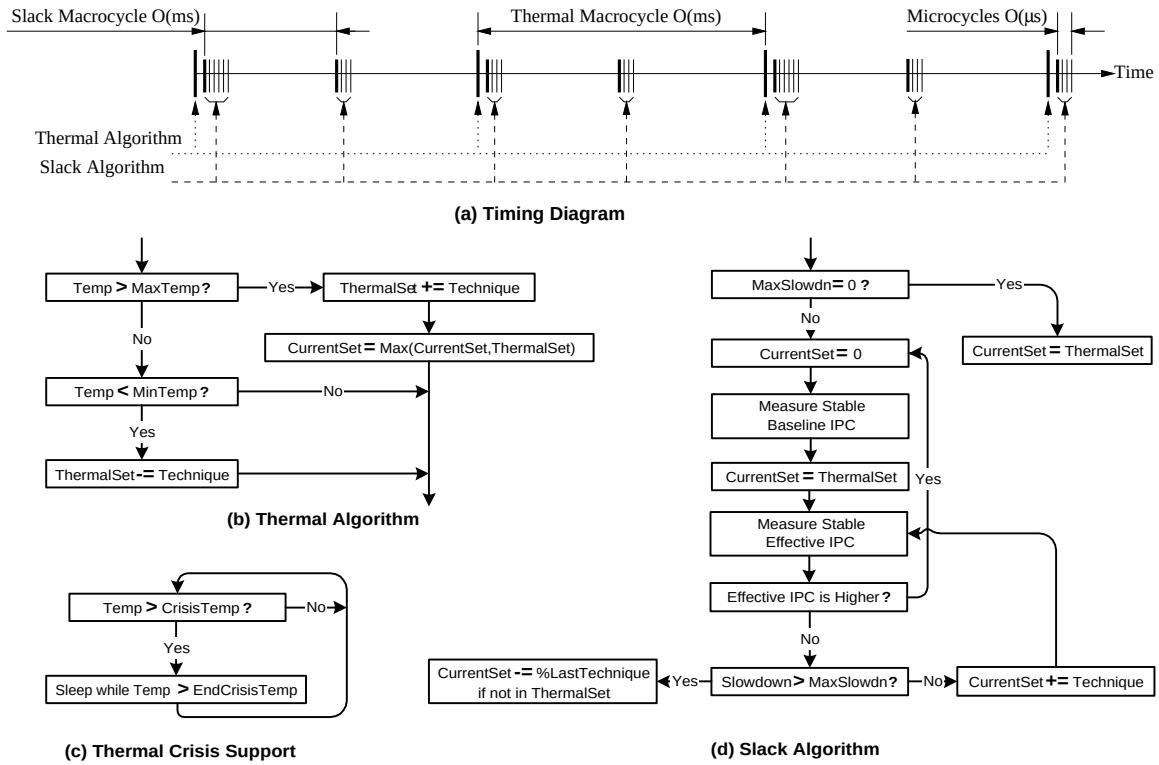


Figure 1: Algorithms used in our framework.

we minimize the chances that the deactivation of a technique brings us back to over *MaxTemp*.

Note that, in some cases, we may not be able to prevent the temperature from rising over the limit. For example, such a situation may be caused by a virus. For this reason, the chip must include support for a thermal crisis. One possible such support is shown in Figure 1-(c): if the temperature reaches a *CrisisTemp* temperature, the hardware unconditionally sets the system to sleep until the temperature is safely lower than *CrisisTemp*.

Slack Algorithm

The Slack algorithm is implemented in hardware instead of as an OS routine. The reason is that, every time that it runs, it needs to repeatedly measure the number of instructions executed by the application at μs-level intervals. After several such measurements in the background, the algorithm makes the decision. These intervals we call *Microcycles* (Figure 1-(a)). We will see that, for higher accuracy, a microcycle is of the order of a few μs.

The Slack algorithm is shown in Figure 1-(d). If no slowdown can be tolerated, the Current Set is simply set to the Thermal Set. Otherwise, the Current Set is deactivated so that the hardware can measure the stable baseline IPC of the application. To compute the IPC, the hardware reads at microcycle intervals the counter of instructions executed. It may take several readings until a reasonably stable IPC is obtained. Note that by deactivating all techniques for several μs we do not risk a dangerous temperature surge because the time is too short.

We then set the Current Set to the Thermal Set and, to find out the resulting slowdown, calculate the new *effective* IPC. The new effective IPC is the new measured IPC plus a correction if the Thermal Set includes techniques that change the clock frequency.

With this new effective IPC, we can compare the slowdown caused by setting the Current Set to the Thermal one, to the maximum tolerable slowdown (*MaxSlowdn*). If *MaxSlowdn* is higher, we augment the Current Set with the next highest-priority technique not yet in it and again measure the effective IPC. This process is repeated until the application slowdown is equal to or higher than *MaxSlowdn*. If the slowdown is higher than *MaxSlowdn*, the last technique that has been added to the Current Set is marked as active for only a fraction of the Slack macrocycle, such that the final slowdown ends up being no higher than *MaxSlowdn*. The only exception is when this last technique added belongs to the Thermal Set, in which case, it cannot be deactivated. Finally, when we reach this point, the algorithm exits.

Every time that we go through the loop of adding a new technique to the Current Set, the hardware may need to take several measurements spaced one microcycle apart, until a stable IPC is obtained. Unfortunately, it is possible that, at the same time, the application also goes through a change in its regime that induces a change in IPC. In this case, to avoid confusing our algorithm, we proceed as follows. If the effective IPC suddenly becomes higher after activating a technique, it is clear that the regime changed. If we pressed on with more techniques until we reached the original target IPC, we would be slowing down the application beyond the tolerable limit. Consequently, as shown in Figure 1-(d), we stop the algorithm and restart it from the beginning.

If, instead, the regime change is in the opposite direction, our algorithm will not notice it: we will assume that the technique just activated is solely responsible for the large IPC reduction. However, this is fine. Our algorithm will end up producing a conservative solution: in the final system, the true slowdown relative to the baseline execution will be less than it could be tolerated. Consequently, the end user is not negatively affected.

Note that some of the techniques used may have non-trivial activation delays. Such is the case, for example, for voltage-frequency scaling, which takes 10-20 μs to activate or deactivate [11]. Such delays, however, are negligible compared to the duration of a macrocycle. For example, if a slack macrocycle takes 2 ms, activating and deactivating voltage-frequency scaling takes only about 2% of the macrocycle. Furthermore, because the impact of voltage-frequency scaling on the IPC is fairly predictable, we do not need to deactivate it at every beginning of a macrocycle to estimate the baseline IPC. This fact further reduces overhead.

Finally, since both the Thermal and the Slack algorithms may update the Current Set, we need to prevent inconsistencies. To this end, and also to ensure that slack macrocycles are not cut off short, we propose the following timing (Figure 1-(a)). We choose the slack macrocycle so that a thermal one contains several slack macrocycles plus a few μs. After the OS has executed the Thermal algorithm and is about to return execution to user mode, it sets the hardware to trigger the next run of the Slack algorithm in a few μs. We set this delay so that, when the Slack algorithm finally runs, it finds the user application in a warmed-up state. From then on, the Slack algorithm runs periodically, always in the background. Finally, when an interrupt triggers the Thermal algorithm again, the first action of the OS is to temporarily disable the hardware that triggers the Slack algorithm. If it so happens that the Slack algorithm was running at the time, this action stops it and automatically sets the Current Set to the Thermal Set.

2.3 Software Interface

The *MaxTemp* and *MaxSlowdn* registers presented above are part of our framework's software interface. In addition, for each energy-management technique, the interface contains a register with the relative priority of activation of the technique (Figure 2). All registers are set by the OS, although *MaxSlowdn* can also be set by the application. With this support, our algorithms can decide what techniques to include at any time in the Current Set.

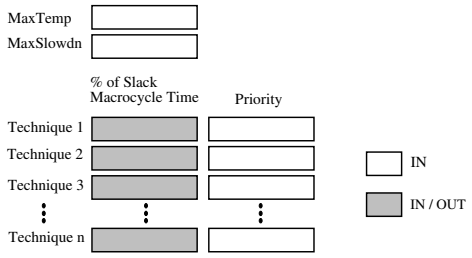


Figure 2: Software interface of our framework.

However, the OS should also have a means to directly overwrite the decisions taken by our default algorithms. This capability can be useful when the OS has specific information on the performance or energy characteristics of the application that is running. Such information may be available from a profile of the application.

One way to extend the interface is to allow the OS to overwrite the decisions of the algorithms as shown in Figure 2. We add one input/output register for each technique in the framework. For a given technique, the register indicates the fraction of the slack macrocycle for which the technique is activated. While these registers are automatically set by the Slack algorithm as it adds techniques to the Current Set, they can also be overwritten by the OS.

2.4 Related Issues

Two important related issues are whether to implement the algorithms in hardware or in software, and whether to make the decisions in a centralized or distributed manner in the chip. We consider these issues next.

2.4.1 Hardware vs Software Implementation

The Thermal algorithm is implemented as an OS interrupt handler. While the Slack algorithm could also be implemented in software, we choose to implement it in hardware. This is in contrast to related algorithms proposed in the literature that exploit system idleness in software [4, 25].

A software implementation of the Slack algorithm would certainly be sufficient if we restricted our work to a certain class of energy-management techniques or to a certain class of applications. Specifically, suppose that we restricted our techniques to those that induce predictable slowdowns like voltage-frequency scaling. In this case, the OS can simply activate the technique for the time duration that will induce the desired slowdown.

Likewise, software might be enough if we restricted the applications to those that, by repeating certain high-level operations, easily tell how fast they are executing. For example, consider video streaming applications. Their speed can be easily monitored by recording the number of frames per unit of time that are being processed. It is easy for the OS to know what is the slowdown caused by a certain energy-management technique by simply checking the new frame rate. There is no need to measure the IPC.

However, we want our Slack algorithm to deliver accurate solutions for all classes of techniques and applications. To see why it requires a hardware implementation, recall that the Slack algorithm repeatedly measures the IPC of the application. While software can support measurements at ms-level intervals, only a hardware solution can support measurements at μs-level intervals. In practice, we need a hardware solution only if the behavior of the application changes significantly at ms-level intervals while staying relatively uniform at μs-level intervals.

We have evidence that μs-level measurements are beneficial in our applications. To understand why, consider a loop. In general, IPC measurements at μs-level intervals will yield fairly uniform values, irrespective of the duration of the loop, as long as 1 μs includes a few iterations. However, IPC measurements at ms-level intervals will yield uniform values only if the loop lasts for many ms. In our applications, much of the code appears to exhibit more uniformity at μs-level intervals than at ms-level intervals. Consequently, we set the interval between measurements (microcycle) to a few μs and, therefore, implement the Slack algorithm in hardware.

2.4.2 Distribution vs Centralization

We now consider how to apply our framework to chips with multiple processor cores. Ideally, we would like to run the framework in a distributed manner. Each processor would have its own framework, running algorithms that read local sensors and make decisions on what techniques to activate locally. This approach is appealing because, potentially, each processor may be running a very different application.

In practice, while some energy-management techniques like those that modify the cache hierarchy can be easily controlled on a per-processor basis, other techniques are best controlled for the whole chip. Consider, for example, voltage-frequency scaling. Using a different voltage and frequency in each processor neighborhood introduces complexity and makes communication between the processors trickier.

One possible alternative is to use per-processor frameworks to run the algorithms and then, after a global synchronization step, make a global decision. However, such an approach is likely to suffer from synchronization overhead.

The approach that we take is to run the algorithm in a centralized manner. Signals from the different processor neighborhoods bring information from the distributed sensors to a central framework module. The module feeds the highest temperature and the sum of all the instructions executed to a centralized algorithm. While this approach requires a more careful timing design, it simplifies the decision-making process.

2.5 Energy Management Techniques

The different energy-management techniques in the framework will target different components of the chip and impact the energy, execution time (delay), and energy-delay product of applications differently. In this section, we select a few, representative techniques to include in the prototype framework that will be evaluated in Section 4.

All the techniques that we select reduce the average power consumption at the expense of slowing down the application. However, while some techniques reduce the total energy consumed in the application run, others do not. Consequently, the techniques in the first group may or may not decrease the energy-delay product, while those in the second group always increase it.

Among the techniques in the first group, we include: sub-banked data caches [9, 30], filter instruction caches [20], voltage-frequency scaling [11, 13], and reduced memory voltage [16]. In each of these cases, when the technique is activated, the system goes from a default configuration to a lower-energy, lower-performance one. These techniques can be used for both the Thermal and Slack algorithms.

Among the techniques in the second group, we include slowing down data cache hits and putting the processor to light sleep. These techniques simply introduce extra delay to reduce the average power. Due to their energy inefficiency, we will try to keep them out of our Thermal and Slack algorithms. However, they may contribute to the thermal crisis support.

We now briefly describe these techniques, while a more detailed description can be found in [36]. The values used for their parameters are listed in Section 3.1. Our framework can be easily extended to include other techniques.

Sub-Banked Data Cache

With cache sub-banking, a cache access activates only part of the cache line selected instead of the whole line [9, 30]. To support sub-banking, the cache is augmented with additional decoding logic and transmission gates. When sub-banking is not activated, this logic adds negligible delay to the cache access time.

When sub-banking is activated, a cache access consumes less energy. This is because the number of activated bit lines and sense amplifiers is reduced. However, the presence of the extra decoding logic and transmission gates tends to increase the cache access time. Consequently, cache hits consume less energy but are slower. The energy consumption and speed of cache misses are unaffected.

Filter Instruction Cache

The on-chip I-memories that supply instructions to the processors in an embedded chip are often designed with high-performance SRAM to ensure that their latency is minimal. They are also large, to hold the whole program. As a result, each access to them, while fast, consumes significant energy.

To address this problem, a small I-cache can be placed between the I-memory and the processor. Accesses to this cache are not faster in number of cycles than accesses to the already fast I-memory. However, they consume much less energy. As a result, this cache works somewhat like a filter cache [20].

If this filter cache is deactivated, all fetches go directly to memory, enabling a fast yet energy-consuming system. If, instead, the cache is activated, hits in the cache take the same time but consume much less energy. Misses, however, force the fetch to go to memory, adding up additional latency and energy consumption. Overall, with the cache activated, the system is likely to be slower but consume less energy.

An alternative design could be to eliminate the filter cache and add sub-banking to the I-memory. In such a design, however, accesses to an I-memory sub-bank could suffer one extra cycle of latency. The result is likely to be a slower system than the one with the filter cache.

Voltage-Frequency Scaling

Reducing both the voltage and the frequency of the chip is a well-known technique [11, 13]. Dynamic energy is proportional to the square of the supply voltage, while dynamic power is proportional to the frequency and to the square of the voltage. To apply this technique, we simply reduce linearly the voltage and frequency of the whole chip to V_{dd_low} and f_{low}. This change works for the linear section of the scaling curve.

Reduced Memory Voltage

We lower the voltage of only the DRAM array to V_{mem_low}. This can be done by changing the reference voltage used in an on-chip voltage converter according to the outputs of a detector [16]. Voltage changes have to be managed carefully because they induce non-linear changes to transistor characteristics. In this technique, to scale down other parameters as we scale down the voltage, we use circuit simulations. In addition, during the low-power mode, we also change the DRAM refresh intervals. The procedure that we use is outlined in [36].

Slowing Down Data Cache Hits

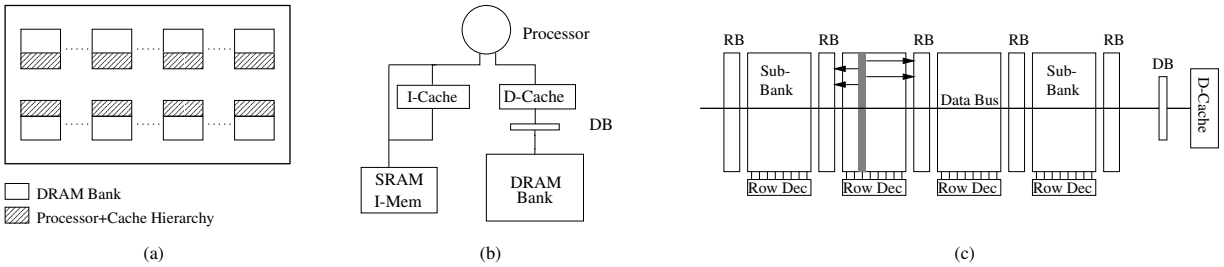
This technique progressively reduces the number of outstanding data loads and stores that a processor can have and, later, increases the latency of cache hits. More specifically, the number of allowed outstanding accesses is progressively halved. Once we reach 1 load and 1 store, we progressively increase the cache hit latency one cycle at a time. When this technique is to be deactivated, we undo these changes in reverse order.

Light Sleep Mode

In this technique, we put the processor in a light sleep mode for a period of time. We do not turn off the PLL, clock distribution, or DLLs to minimize any wake-up penalty. We simply gate the clock at the output of the DLLs. Since, by default, we were already clock-gating all the units not used, this technique cannot save much energy. In fact, because we are keeping the PLL, DLLs, and clock distribution lines on while slowing down the application, this technique ends up increasing the energy consumed. However, it reduces the average power consumed in the system.

3 EVALUATION ENVIRONMENT

We evaluate an implementation of our adaptive framework on top of an advanced chip with multiple superscalar cores and DRAM banks. We use detailed software simulations at the architectural level. The simulations are performed using



(a) (b) (c)

Figure 3: Chip architecture modeled: overview of the chip (a), per-processor memory hierarchy (b), and per-processor DRAM bank organization (c). In the charts, *RB*, *DB*, and *Row Dec* stand for row buffer, data buffer, and row decoder, respectively.

Processor	D-Cache	I-Cache	I-Memory	Data Buffer	Row Buffer	DRAM Sub-Bank
2-issue in-order at 800 MHz	Size: 8 KB	Size: 128 inst.	Size: 8 KB	Number: 1	Number: 5	Number: 4
BR Penalty: 2 cycles	Assoc: 2	Assoc: 1	Line: 4 inst.	Size: 256 b	Size: 1 KB	Num Cols: 4096
Int,Ld/St,FP Units: 2,1,0	Line: 32 B	Line: 4 inst.	RTrip: 1.25 ns	Bus: 256 b	Bus: 256 b	Num Rows: 512
Pending Ld,St: 2,2	RTrip: 1.25 ns	RTrip: 1.25 ns		RTrip: 3.75 ns	RTrip: 7.5 ns	RTrip: 15 ns

Table 1: Parameters for a single memory bank and processor pair. In the table, *BR* and *RTrip* stand for branch and contention-free round-trip latency from the processor, respectively.

Technique	Label	Parameter Value
Sub-banked data cache	*SubBank*	Cache hit if no sub-banking: RTrip = 1.25 ns, E = 222.8 pJ Cache hit if sub-banking: RTrip = 2.50 ns, E = 69.1 pJ
Filter instruction cache	*IFilter*	I-mem access: RTrip = 1.25 ns, E/inst = 51.6 pJ I-cache hit: RTrip = 1.25 ns, E/inst = 15.4 pJ I-cache miss + I-mem access: RTrip = 2.5 ns, E/inst = 67.0 pJ
Voltage-freq. scaling	*VoltFreq*	V_{dd_low} = 1.44 V, f_{low} = 640 MHz, overhead of any scaling = 10 μs
Reduced memory voltage	*MemVolt*	V_{dd} = 1.8 V: RB access (RTrip = 7.5 ns, E = 500.1 pJ), DRAM access (RTrip = 15 ns, E = 3702.2 pJ) V_{dd} = 1.2 V: RB access (RTrip = 7.5 ns, E = 500.1 pJ), DRAM access (RTrip = 21.25 ns, E = 2634.6 pJ)
Slowing D-cache hits	*SloHit*	–
Light sleep mode	*Sleep*	–

Table 2: Values of the parameters used in our energy-management techniques. In the table, *E*, *RB*, and *RTrip* stand for energy, row buffer, and contention-free round-trip latency from the processor, respectively.

a MINT-based [32] execution-driven simulation system [21] that models all the components of the chip, including the superscalar processors. The simulator includes energy consumption models. In the following, we describe the architecture modeled, how we estimate the energy consumed, the applications executed, and the metrics used.

3.1 Architecture Modeled

As an example of an advanced chip, we model a processor-in-memory chip with 64 simple processors cycling at 800 MHz and 64 Mbytes of DRAM. The target technology is IBM's 0.18 μm Blue Logic SA-27E ASIC [12] with some expected improvements in DRAM density [36]. The default voltage is 1.8 V.

The chip is modeled after a *FlexRAM* chip [19]. Processors are 2-issue wide and statically scheduled. Each processor is associated with a 1-Mbyte DRAM bank. A processor can directly access its own DRAM bank as well as the DRAM of its left and right neighbors. Such support allows communication between the processors, effectively connecting them in a ring. In addition, as in *FlexRAM*, the chip contains an on-chip controller that executes the serial sections of the application, including initialization, broadcast, and reduction operations [19]. The controller's contribution to the execution of our applications constitutes on average only 8% of the time, and is mostly limited to the initialization and ending parts of the application. For these reasons and because most chip resources are very underutilized when the controller runs, we do not include the controller's contribution in our evaluation.

Figure 3 shows the architecture of the chip. In the figure, Chart (a) gives an overview of the chip, while Chart (b) shows the memory hierarchy of each processor in the chip and Chart (c) shows the organization of each DRAM bank into sub-banks. Table 1 shows the most important architectural parameters for a single memory bank and processor pair.

Table 2 shows the values for the parameters of the energy-management techniques included in our framework. The energy values used will be justified in the next section. The values of some other framework parameters are as follows. Changing the memory voltage with *MemVolt* is assumed to have negligible overheard. Both the thermal and the slack macrocycles are set to 1 ms, while the microcycle is set to 1 μs. To avoid instability in the Thermal algorithm, we set a different *MinTemp* for each technique, as shown in Section 3.4. Finally, every time that we execute the Thermal algorithm, we charge 200 cycles to account for the overhead of the execution in the OS.

3.2 Estimating the Energy Consumed

To estimate the energy consumed in the chip, we have applied scaling-down theory to data on existing devices reported in the literature, as well as used several techniques and formulas reported in the literature [3, 30, 18, 24, 34, 35]. A detailed discussion of the methods that we have followed can be found in [36]. In this section, we give an overview of how we estimate the energy consumed in the processor cores, memory hierarchies, and clocks. We also discuss how we validated the models.

Processor Cores

Each core is a 32-bit 2-issue processor with a DLX-like pipeline. It supports a simplified version of the MIPS ISA with only 28 16-bit instructions [19]. We take the data from [35] and, by applying general scaling theory and considering technology trends, we estimate the average energy consumed in the register file, branch unit, ALU, and the other modules of the processor. Then, we can estimate the energy consumed by each type of instruction by adding up the energy of all the modules used by that particular instruction type. We assume perfect clock gating inside the processor code. With this approach, for example, we estimate that an add, a branch, and a multiply instruction consume an average of 56.1, 34.8, and 251.2 pJ, respectively.

Memory Hierarchies

To compute the energy consumed in the memory hierarchy, we use popular models [30, 18]. We classify memory hierarchy accesses based on what level of the hierarchy they reach, and depending on whether they are reads, writes, or dirty line displacements. Then, we compute the average energy consumed by one access of each class. This is done by dividing the access into simple operations. For example, a read that hits in the row buffer is divided into a cache tag check, a read hit in the row buffer, and a line fill into the cache. Finally, to compute the overall energy in the memory hierarchy, we multiply the number of accesses of each class times the corresponding energy per access in the class, and then accumulate the contribution of all classes. As an example, Table 3 shows the average energy consumed by a read and a write access to different levels of the hierarchy.

Level of the Hierarchy	Rd Energy (pJ)	Wr Energy (pJ)
D-cache	222.8	246.3
I-mem (per instr)	51.6	56.8
Row buffer	500.1	2740.6
DRAM bank	3702.2	3286.2

Table 3: Average energy consumption per access.

Clocks & Other

The clocking system includes 1 main PLL and 16 distributed local DLLs [29]. The clock network is laid out in the chip using an H-tree structure to minimize skew. To estimate the overall energy of the clocking system, we estimate and add the contributions of several components, namely PLL, DLLs, buffers, and distribution lines. Such contributions are estimated based on [3] and on capacitance models. Overall, the estimated average energy per cycle is 957.5 pJ. This figure does not include the energy for the clock inside the processor cores. The latter is included in the computation for the cores. Further details can be found in [36].

Validation

We validate our energy estimates with several experiments. We report on two of them here. In the first validation, we examine our cache model. We compare our energy estimates to those generated with the CACTI v2 models [34]. Since CACTI uses a relatively old sense amplifier model, we change it to a more aggressive one. The comparison shows that our estimates of energy consumption in the data cache and CACTI's are only 9% different [36].

In a second validation, we focus on the relative energy consumption of the I-cache, D-cache, clock, and processor core. Such a relative breakdown of energy for the Strong ARM processor is available from [24]. We compute the corresponding estimates for one of our processors plus its associated caches and share of the clock. While there are some differences between the two architectures, getting a similar breakdown is reassuring. The comparison shows that the contribution of each of the components does not differ by more than an absolute 6% between the two systems [36].

3.3 Applications Executed

For the experiments, we use 6 applications that are suitable to the integer-based processor-in-memory chip considered: they access a large memory size, are very parallel, and are integer based. They come from several industrial sources. We have parallelized each application into 64 threads by hand.

Table 4 lists the applications and their characteristics. They include the domains of data mining, neural networks, protein matching, multimedia, and image compression. Each application runs for several billions of instructions. Appendix A gives more information on each application.

3.4 Metrics Used

We characterize an application run with four metrics: performance (measured with total execution time, also called *delay*), average power consumption, total energy consumption, and product of energy times execution time (energy-delay product [10]). We will strive for a low energy-delay product, since it implies a good balance between high speed and low energy consumption.

In some experiments, we need to estimate chip temperature. However, our models only use energy and power metrics. We currently do not have a thermal model that, taking into account the chip package and cooling support, translates sustained power dissipation into chip temperature.

It is known, however, that heat transfers occur at the ms level [31]. As a result, it has been suggested to use the average power dissipated over many cycles as a proxy for temperature [5]. We follow this approach and use a metric called $Power^*$ as a proxy for chip temperature. At a given time, $Power^*$ is 0.75 times the average power consumed by the chip in the last millisecond plus 0.25 times the value of $Power^*$ a millisecond ago. While clearly not perfect, this recursive definition tries to approximate the behavior of temperature. Using this metric, the proxy for *MinTemp* for *VoltFreq*, *SubBank*, and *IFilter* is set to 45%, 75%, and 78%, respectively of the proxy for *MaxTemp*.

4 EVALUATING THE FRAMEWORK

To assess our DEETM framework, we evaluate three issues: the management of multiple energy-management techniques (Section 4.1), the Thermal algorithm (Section 4.2), and the Slack algorithm (Section 4.3).

4.1 Technique Analysis & Comparison

Given a DEETM framework with multiple techniques, the first question to ask is what combination of techniques should it apply and in what order. We now answer this question for our framework.

Comparing Individual Techniques

We start by comparing the individual techniques with the following experiment for each application. We execute the application without activating any technique and record the average power dissipated P_{orig} (last column of Table 4). Then, for each technique, we perform four runs dynamically activating the technique with different intensities. The inten-

Appl.	What It Does	Problem Size	D-Cache Hit Rate	Average Power(W)
GTree	Data mining: tree generation	5 MB database, 77.9 K records, 29 attributes/record	0.507	10.2
DTree	Data mining: tree deployment	1.5 MB database, 17.4 K records, 29 attributes/record	0.986	10.8
BSOM	BSOM neural network	2 K entries, 104 dimensions, 2 iterations, 16-node network, 832 KB database	0.947	15.5
BLAST	BLAST protein matching	12.3 K sequences, 4.1 MB total, 1 query of 317 bytes	0.969	8.7
Mpeg	MPEG-2 motion estimation	1 1024x256-pixel frame plus a reference frame. Total 512 KB	0.999	11.3
FIC	Fractal image compressor	1 512x512-pixel image, 4 512x512-pixel internal data structure. Total 2 MB	0.978	6.1

Table 4: Applications executed.

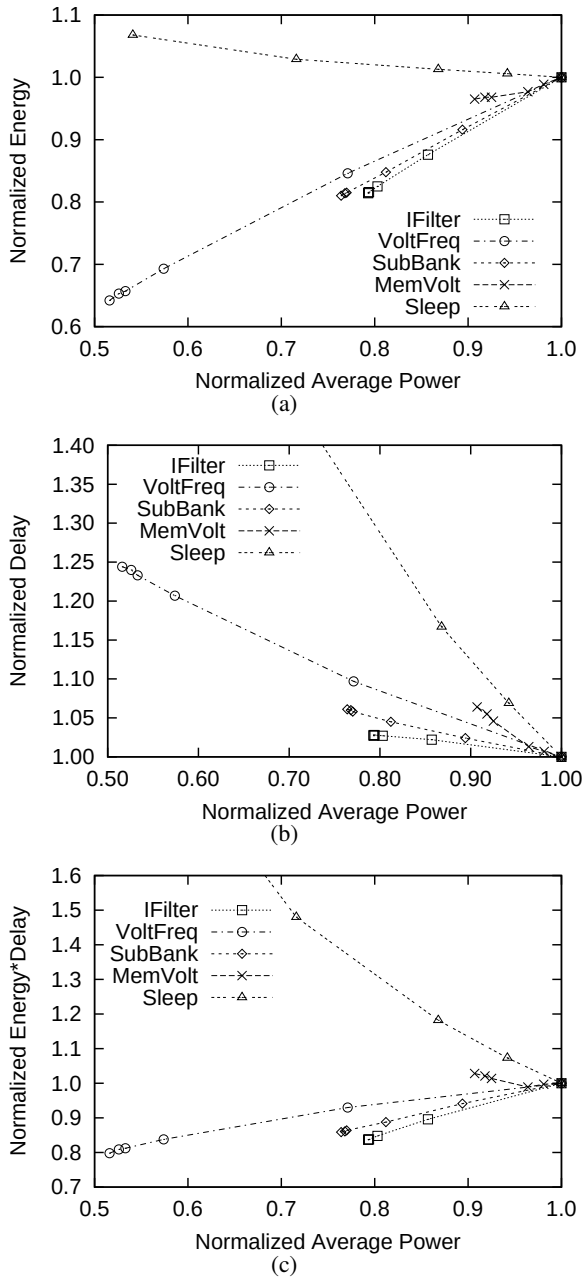


Figure 4: Impact of dynamically applying each individual energy-management technique: total energy consumed by the applications (a), their execution time (b), and their energy-delay product (c). The data is normalized to a run with no active technique and then averaged out across all applications.

sity is regulated with a power threshold: if the power in the last microcycle was over the threshold, the technique gets activated; the technique is deactivated when the power in the last microcycle was such that the technique could be deactivated without going over the threshold again. We set the thresholds to $1.2 \times P_{orig}$, $1.0 \times P_{orig}$, $0.8 \times P_{orig}$, and $0.6 \times P_{orig}$. Finally, we perform an experiment activating the technique for the whole run.

Figure 4 shows the results. The results of each run have been normalized to the run with no active technique for the same application, and then averaged out across all applications. The figure shows the resulting average power consumed in the run (X axes) against the total energy consumed (Chart (a)), execution time (Chart (b), where execution time is labeled *Delay*), and energy-delay product (Chart (c)). Since *SloHit* has a behavior very similar to *Sleep*, we do not show *SloHit* to simplify the charts.

The figure shows that the behavior of *Sleep* is different from the others as the average power decreases. *Sleep* does not reduce the energy (Chart (a)), substantially slows down the applications (Chart (b)) and, as a result, increases the energy-delay product significantly (Chart (c)). Consequently, due to its inefficiency, we only use it as the last resort in a thermal crisis.

The other four techniques (*IFilter*, *SubBank*, *VoltFreq*, and *MemVolt*) decrease the energy consumed by the chip (Chart (a)) and, while they still slow down the application (Chart (b)), they manage to reduce the energy-delay product or keep it roughly constant (Chart (c)). They differ significantly, however, in the slope of their curves and in the maximum power reduction that they can deliver. The maximum reduction is delivered when they are applied statically. This situation corresponds to the leftmost point of each curve.

To compare these four techniques to each other, we examine Chart (c). Recall that we want to minimize the energy-delay products. Under this requirement, the chart tells us what is the best technique to apply individually, and how to rank the techniques in case we want to apply them in a combined manner.

If we want to apply a single technique, we should choose the one that, for the desired average power reduction, delivers the lowest energy-delay product. For example, for power reductions that are less than 20%, *IFilter* is the best. *SubBank* is the best if we want reductions between 20 and 25%, while *VoltFreq* is the best for reductions larger than 25%. From this data, we can see that *IFilter* and *SubBank* are good but limited. Since their scope is only memory system accesses, they deliver modest power reductions.

If, instead, we want to rank the techniques for a possible combined application of them, what matters is not the absolute power reduction but the slope of the curves. Specifically, we approximate each curve with a straight line and record the slope of the line. The techniques with the highest positive slopes should be given the highest priority. Consequently, in our framework, the order of application of the techniques, irrespective of the power reduction desired, should be *IFilter*,

then *SubBank*, then *VoltFreq*, and so on.

Note that, for our techniques, the shape of the curves makes it possible to reasonably approximate each curve with a single straight line. This may not be true, however, in other scenarios, where we would need different straight lines in different segments of a given curve. In this case, the ranking of techniques would not be as straightforward: it would depend on the power reduction desired.

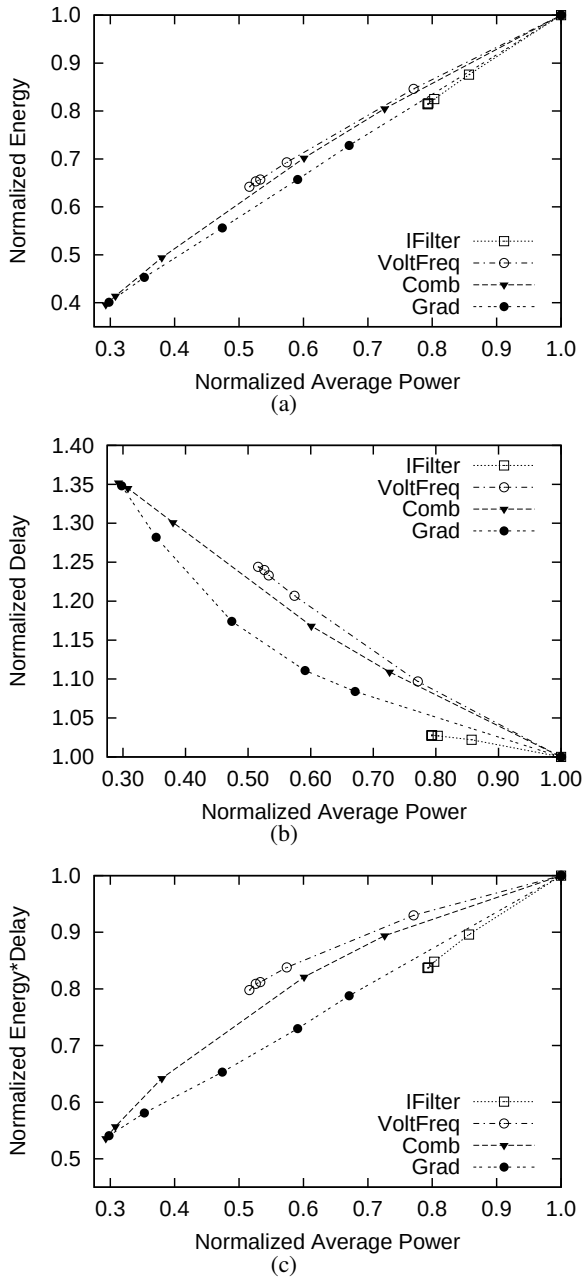


Figure 5: Impact of dynamically applying a combination of energy-management techniques: total energy consumed by the applications (a), their execution time (b), and their energy-delay product (c). The data is organized as in Figure 4.

Another complication occurs if the slope of a curve changes when the technique is combined with other techniques. While we have observed this effect in our framework, it does not change the ranking of techniques listed above.

Finally, we note that *MemVolt* reduces neither the average power much nor the energy-delay product. It is, therefore, unattractive. Its scope for impact is limited to applications with many cache misses. Unfortunately, even in this case, we find that it works poorly because the slower DRAM becomes a contention bottleneck that slows down the application (Chart (b)).

Applying Combined Schemes

To see the potential of our framework, we combine the three most effective techniques, namely *IFilter*, *SubBank*, and *VoltFreq*, into a single scheme. We consider two different schemes: *Comb* activates and deactivates the three techniques simultaneously, while *Grad* activates and deactivates them gradually. *Grad* uses the ranking selected before: it activates *IFilter* first; if more power or energy reduction is needed, it activates *SubBank*; if more is needed, it activates *VoltFreq*. When the techniques must be deactivated, it follows the reverse order.

Figure 5 shows the results of repeating the experiments of Figure 4 for *Comb* and *Grad*. For reference purposes, the figure also includes the curves for *VoltFreq* and *IFilter* from Figure 4. Note, however, that the axes have been expanded relative to Figure 4.

We can see from Figure 5 that, for modest power reductions, the effectiveness of *Comb* is between that of *VoltFreq* and *IFilter*. Specifically, Chart (c) shows that, for a given power reduction, the energy-delay product of *Comb* is between that of *VoltFreq* and *IFilter*. Consequently, *Comb* works well. In addition, *Comb* can deliver much higher power reductions than the individual techniques: if *Comb* is statically applied, it can reduce the average power by up to 70%. As a result, the final energy-delay product obtained in Chart (c) is also much lower than for the individual techniques.

As can be seen in the figure, however, *Grad* is better. Chart (c) shows that, for modest power reductions, this scheme delivers energy-delay products that are nearly as low as *IFilter*, the best of the three techniques. This is because, for this range of reductions, *Grad* is largely *IFilter*. When larger reductions are desired, *Grad* starts using the less optimal techniques. Finally, as we approach large reductions, it gets closer to *Comb*. In all cases except static application, however, *Grad* has a lower energy-delay product than *Comb* (Chart (c)).

These results form the rationale behind our choice of Thermal and Slack algorithms in Section 2.2: a gradual, priority-ordered application of techniques that reduce the energy-delay product. Consequently, we implement the Slack and Thermal algorithms with *Grad*. In addition, as part of the Thermal algorithm, we keep one additional technique ready for activation in case of a thermal crisis. Such a technique, which must be able to reduce the average power consumed as much as needed, is chosen to be *Sleep*.

Variation Across Applications

Finally, we note that, although different applications behave differently, the schemes chosen for our adaptive framework work well across all applications. For lack of space, we only briefly discuss the two individual applications that diverge the most from the average: *GTree* and *DTree*. *GTree* has a high data cache miss rate (Table 4), which causes *SubBank* to have relatively less impact. *DTree*, on the other hand, has relatively more I-cache misses, which causes *IFilter* to be less effective. Overall, however, it can be shown that *Grad* is very effective: it reduces the energy-delay product significantly, while enabling large reductions in average power.

4.2 Evaluating the Thermal Algorithm

The goal of the Thermal algorithm is to keep the temperature of the chip lower than *MaxTemp*, while minimizing any resulting application slowdown. In addition, under no condition should the temperature surpass *CrisisTemp*. As indicated before, we use *Grad* and, if *CrisisTemp* is reached, we activate *Sleep*. We call the resulting scheme *Grad+Sleep*.

To show that *Grad+Sleep* is effective, we demonstrate that, given different *MaxTemp* temperature limits, it effectively keeps the chip temperature below *MaxTemp* practically all the time, while slowing down the execution only modestly. Recall that, as stated in Section 3.4, we use $Power^*$ as a proxy for temperature.

In Figure 6, we show the results of applying *Grad+Sleep* under different $Power^*$ limits. These limits are proxies for *MaxTemp*. For each application, the limits considered are $1.2 \times P_{orig}$, $1.0 \times P_{orig}$, $0.8 \times P_{orig}$, and $0.6 \times P_{orig}$, where P_{orig} is the original average power of the application (last column of Table 4). To get an idea of the absolute values of these limits, if we average them out across all the applications, we get 12.5, 10.4, 8.3, and 6.3 W, respectively. The crisis $Power^*$ is set sufficiently high such that it is never reached. As usual, the data is normalized to the original conditions of the application and then averaged out across all applications.

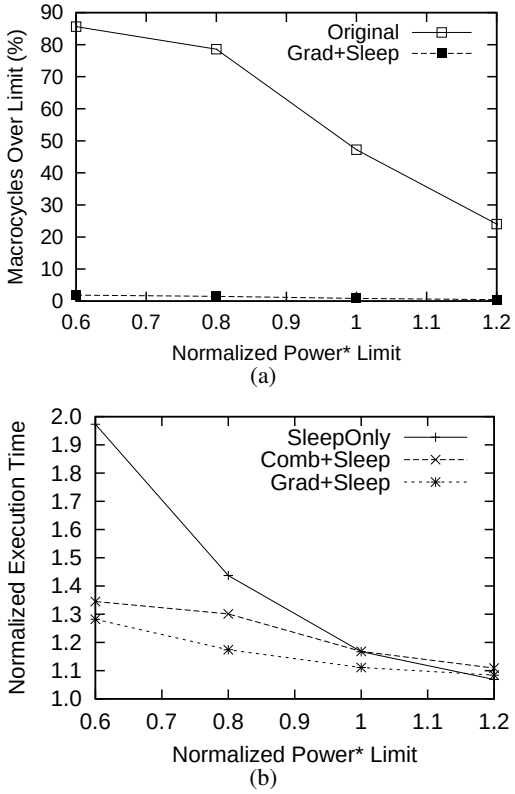


Figure 6: Impact of enforcing different $Power^*$ limits in the chip: fraction of thermal macrocycles over the $Power^*$ limit (a) and resulting execution time of the applications (b). These limits are proxies for *MaxTemp*.

Figure 6-(a) shows the fraction of thermal macrocycles where $Power^*$ is above the limit before we activate our framework (*Original*) and after (*Grad+Sleep*). The chart shows that, irrespective of how low we set the limit to, our framework keeps $Power^*$ below it for practically all the time. This is true even after setting the limit to 0.6 times the average power in the chip before activating the framework, which is the leftmost point of the chart. Such a limit places 85% of the *Original* macrocycles over the limit.

Figure 6-(b) shows the resulting execution time of the applications after activating the framework. If we focus on the *Grad+Sleep* curve, we see that, for modest limits, the scheme induces minimal slowdowns. For example, after setting the limit to 1.2 times the original average power, our framework only slows down the applications, on average, by 8%.

Overall, from the previous two discussions, we see that the goal of the Thermal algorithm is realized. For comparison purposes, however, Figure 6-(b) also shows the impact of using less efficient schemes. *Comb+Sleep* uses *Comb* instead of *Grad*. *SleepOnly* simply uses the *Sleep* technique when $Power^*$ surpasses the limit. More specifically, when a thermal macrocycle records a $Power^*$ higher than the limit, the fraction of non-sleeping cycles in the next macrocycle is decreased proportionally to how much $Power^*$ was over the limit. This scheme is, therefore, self-regulating. From the figure, we see that such schemes induce higher slowdowns than *Grad+Sleep*. *SleepOnly* is especially inefficient for relatively low $Power^*$ limits. However, it works well for the highest limit because it is being applied in a fine-grained manner.

4.3 Evaluating the Slack Algorithm

The goal of the Slack algorithm is to save as much energy as possible without extending the execution of the application beyond a given tolerable slack. As indicated before, we implement the algorithm with *Grad*. To show that our framework is effective, we demonstrate that, given different slack sizes, *Grad* delivers large energy savings without slowing down the job noticeably more than tolerable.

In Figure 7, the framework is tested with different slack sizes, specified as a percentage of the original execution time of the application. As usual, the data is normalized to the original conditions of the application and then averaged out across all applications.

Figure 7-(a) shows the resulting energy consumed by the applications for different slack sizes. The chart shows that *Grad* delivers large energy savings by exploiting even small slacks. For example, if the applications are allowed to exploit a 10% slack, they consume only 60% of the original energy; if they are given a 30% slack, they consume only 40%.

To put the effectiveness of *Grad* in perspective, the chart also shows the curves for $E \times D = constant$ and $E \times D^2 = constant$. As a reference, the voltage-frequency scaling technique [11, 13] often falls in between the $E \times D$ and $E \times D^2$ curves. Indeed, if the scaling of voltage and frequency is linear, since energy is proportional to the square of the voltage and delay is inversely proportional to the frequency, $E \times D^2$ remains constant. In practice, the scaling deviates from linear behavior and we move toward the $E \times D$ curve. Overall, from the distance between these curves and *Grad*, we can see that our framework is very effective, especially with small slacks.

Figure 7-(b) shows the fraction of the tolerable slack that is used up by our framework. We see that, for modest-sized slacks, *Grad* tends to deviate little from using the maximum tolerable slack. Any under- or over-utilization is limited to about 2% of the slack. As the slack increases over 35% of the execution time, the applications cannot use it all, even when all the techniques in *Grad* are in full operation. As a result, part of the slack is wasted. Overall, we see that the goal of the Slack algorithm is realized: *Grad* delivers large energy

reductions by exploiting even small slacks.

To gain insight into any possible improvements over *Grad*, Figures 7-(a) and 7-(b) also show the behavior of an ideal scheme that we call *Oracle*. At any given microcycle in the execution, *Oracle* applies the combination of *IFilter*, *SubBank*, and *VoltFreq* that best furthers the goal of the Slack algorithm. Since *Oracle* is based on perfect knowledge of the future, it should have, for a given slack, the lowest energy curve in Chart (a). In some cases, however, *Grad* reduces the energy slightly more than *Oracle*. This is because, due to imperfect prediction of the future, *Grad* sometimes goes slightly over the tolerable slack in Chart (b). Overall, however, the charts show that there is not much difference between the *Oracle* and *Grad* curves, which suggests that *Grad* is very competitive.

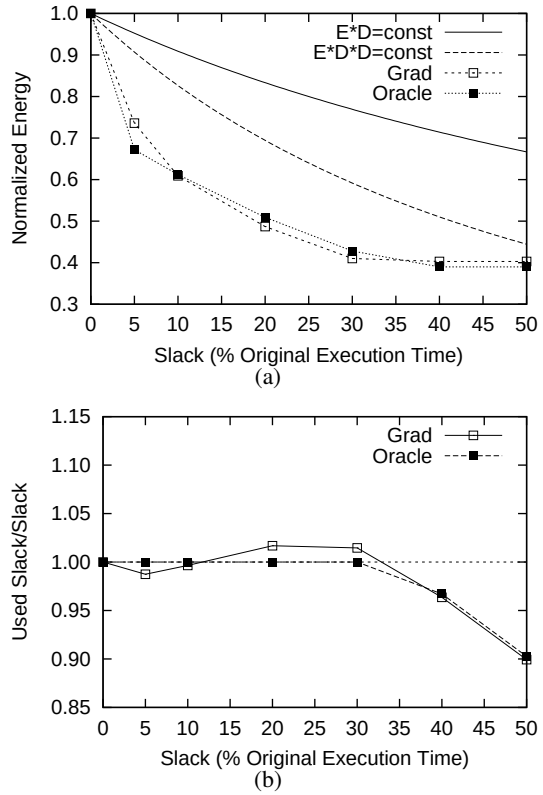


Figure 7: Effect of exploiting different execution slacks: resulting energy consumed by the applications (a) and fraction of the slack that is used up (b).

5 RELATED WORK

Of all the techniques and systems listed in Section 1, the work most related to ours is the one on dynamic systems for chip-level energy management. These systems can be classified into three groups. The first one targets temperature control, for example through context switching to jobs that consume less power [27] or through speculation control [5]. The second group targets energy efficiency without compromising performance, for instance through speculation control [23] or through reconfigurability [1]. A final group targets energy efficiency by exploiting slack and, therefore, slowing down the system. This is done, for example, through voltage and frequency scaling [25] or through switching to less aggressive instruction issue and speculation support [8]. Our work is different in two ways: we target both energy efficiency and temperature control, and we combine many techniques in a unified dynamic framework.

Recently, dynamic application of voltage and frequency scaling or various sleep modes have become popular among microprocessors [11, 13].

A related approach is that of ACPI (Advanced Configuration and Power Interface), an open industry specification that defines an interface for the OS to activate low-power modes [14]. Our work differs from ACPI in two ways. First, in ACPI, any decision and control of power modes is done by the OS. In our framework, the decision and control is best done with a combination of software and hardware, which enables finer-grained energy management. Second, current ACPI releases are only concerned with various sleeping modes, while we combine techniques that trade energy for performance.

ACPI and other OS-driven approaches have been used at the system level to save energy dynamically. For example, it is feasible to save energy by dynamically shutting down unused modules of the system like hard disks or the LAN [4]. Alternatively, the savings can come from dynamically reducing the quality of service to the application [7].

6 CONCLUSIONS AND FUTURE WORK

To address the problem of high energy consumption in current and upcoming chips, several schemes for dynamic energy management have recently been proposed. However, such schemes are still relatively limited and, in addition, tend to tackle only one of the two aspects of energy management: either energy efficiency or temperature control. To address these limitations, this paper has proposed a framework for Dynamic Energy Efficiency and Temperature Management (DEETM). The framework addresses the two aspects of energy management in a unified form. In addition, it combines a suite of energy-management techniques that can be activated individually or in groups according to a given policy.

The evaluation has shown that our framework is very effective, especially when the tolerable slowdowns and temperature limits are modest. In these scenarios, dynamic application of the most fitting techniques in the suite is most cost-effective: temperature limits are enforced with small slowdowns and large energy savings are delivered by exploiting small slacks. For example, the framework delivers a 40% energy reduction with only a 10% application slowdown. Overall, we feel that it makes sense for future advanced chips to include a DEETM framework like ours that combines multiple techniques.

As part of our ongoing work, we are trying to improve our DEETM framework by adding more techniques to it. We can then quantify the complementarity of and the overlap between different techniques.

Another approach that we are exploring is the potential of profiling. We can profile an application and, depending on what are its main energy and performance bottlenecks, tailor the activation of the techniques. Experience with the *Oracle* scheme in Section 4.3, however, suggests that little more can be done for the techniques and applications considered. However, other techniques and applications may behave differently. Finally, we are examining how to tailor the framework for different classes of chips, namely high-end microprocessors, chip multiprocessors, and different types of systems on a chip.

REFERENCES

[1] D. Albonesi. Dynamic IPC/Clock Rate Optimization. In *International Symposium on Computer Architecture*, pages 282–292, July 1998.

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, October 1990.

[3] J. Alvarez et al. A Wide-Bandwidth Low-Voltage PLL for PowerPC Microprocessors. *IEEE Journal on Solid-State Circuits*, 30(4):383–391, April 1995.

[4] L. Benini et al. Monitoring System Activity for OS-Directed Dynamic Power Management. In *International Symposium on Low Power Electronics and Design*, pages 185–190, August 1998.

[5] D. Brooks and M. Martonosi. Adaptive Thermal Management for High-Performance Microprocessors. In *Workshop on Complexity Effective Design*, June 2000.

[6] Y. Fisher. *Fractal Image Compression: Theory and Application*. Springer Verlag, 1995.

[7] J. Flinn and M. Satyanarayanan. Energy-Aware Adaptation for Mobile Applications. In *Symposium on Operating Systems Principles*, pages 48–63, December 1999.

[8] S. Ghiasi, J. Casmira, and D. Grunwald. Using IPC Variation in Workloads with Externally Specified Rates to Reduce Power Consumption. In *Workshop on Complexity-Effective Design*, June 2000.

[9] K. Ghose and M. Kamble. Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-Line Segmentation. In *International Symposium on Low Power Electronics and Design*, pages 70–75, August 1999.

[10] R. Gonzalez and M. Horowitz. Energy Dissipation In General Purpose Microprocessors. *IEEE Journal on Solid-State Circuits*, 31(4):1277–1284, September 1996.

[11] T. Halfhill. Transmeta Breaks x86 Low-Power Barrier. *Microprocessor Report*, 14(2):1,9–18, February 2000.

[12] IBM Microelectronics. Blue Logic SA-27E ASIC. http://www.chips.ibm. com/news/1999/sa27e/sa27e.pdf, February 1999.

[13] Intel. *Pentium III Processor Mobile Module: Mobile Module Connector 2 (MMC-2) Featuring Intel SpeedStep Technology*, 2000.

[14] Intel, Microsoft and Toshiba. *Advanced Confi guration and Power Interface Specifi cation*, 1999.

[15] K. Itoh. Low Power Memory Design. In *Low Power Design Methodologies*, pages 201–251. Kluwer Academic Publisher, 1996.

[16] K. Itoh et al. An Experimental 1Mb DRAM with On-Chip Voltage Limiter. In *ISSCC Digest of Technical Papers*, pages 84–85, February 1981.

[17] T. Juan, T. Lang, and J. Navarro. Reducing TLB Power Requirements. In *International Symposium on Low Power Electronics and Design*, pages 196–201, August 1997.

[18] M. Kamble and K. Ghose. Analytical Energy Dissipation Models for Low Power Caches. In *International Symposium on Low Power Electronics and Design*, pages 143–148, August 1997.

[19] Y. Kang, W. Huang, S. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *International Conference on Computer Design*, pages 192–201, October 1999.

[20] J. Kin, M. Gupta, and W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. *International Symposium on Microarchitecture*, pages 184–193, December 1997.

[21] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 286–293, October 1998.

[22] R. Lawrence, G. Almasi, and H. Rushmeier. A Scalable Parallel Algorithm for Self-Organizing Maps with Applications to Sparse Data Mining Problems. Technical report, IBM, January 1998.

[23] S. Manne, A. Klauser, and D. Grunwald. Pipeline Gating: Speculation Control for Energy Reduction. In *International Symposium on Computer Architecture*, pages 132–141, July 1998.

[24] J. Montanaro et al. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. *IEEE Journal Solid State Circuits*, 31(11):1703–1714, November 1996.

[25] T. Pering, T. Burd, and R. Brodersen. The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. In *International Symposium on Low Power Electronics and Design*, pages 76–81, August 1998.

[26] J. Quinlan. *C4.5 - Programs for Machine Learning*. Morgan Kaufmann, 1993.

[27] E. Rohou and M. Smith. Dynamically Managing Processor Temperature and Power. In *2nd Workshop on Feedback-Directed Optimization*, November 1999.

[28] H. Sanchez et al. Thermal Management System for High Performance PowerPC Microprocessor. In *IEEE Computer Society International Conference*, pages 325–330, February 1997.

[29] S. Sidiropoulos and M. Horowitz. A Semidigital Dual Delay-Locked Loop. *IEEE Journal on Solid-state Circuits*, 32(11):1683–1692, November 1997.

[30] C-L. Su and A. Despain. Cache Design Trade-offs for Power and Performance Optimization: A Case Study. In *International Symposium on Low Power Electronics and Design*, pages 63–68, April 1995.

[31] C-H. Tsai. *Temperature-Aware VLSI Design and Analysis*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, May 2000.

[32] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–207, January 1994.

[33] N. Vijaykrishnan et al. Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower. In *International Symposium on Computer Architecture*, pages 95–106, June 2000.

[34] S. Wilton and N. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal on Solid-State Circuits*, 31(5):677–688, May 1996.

[35] N. Yeung. et al. The Design of a 55SPECint92 RISC Processor under 2W. *ISSCC Digest of Technical Papers*, pages 206–207, February 1994.

[36] S-M. Yoo, J. Renau, M. Huang, and J. Torrellas. FlexRAM Architecture Design Parameters. Technical Report CSRD-1584, Department of Computer Science, University of Illinois at Urbana-Champaign, October 2000. http://iacoma.cs.uiuc.edu/flexram/publications.html.

APPENDIX A: APPLICATIONS USED

This appendix describes the applications used. In the following, we use P.Mem to refer to the on-chip controller in the FlexRAM chip that executes the serial sections of the applications. More information on the applications can be found in [19].

GTree is a data mining application that generates a decision tree given a collection of records that we want to classify [26]. The records are distributed across the processors. The P.Mem decides what attributes to use to split the tree and tells the processors what branch they should examine. The processors process their records.

DTree uses the tree generated in *GTree* to classify a database of records [26]. Each processor has a copy of the decision tree and a portion of the database. Each processor processes its local records sequentially. At the end of the execution, the results are accumulated by P.Mem.

BSOM is a neural network that classifies data [22]. Each processor processes a portion of the input. Then, all processors synchronize, a summary of the partial results is combined and re-distributed, and the process begins again. While the original application used floating point, we have converted the application into fixed point to run on our simulated chip.

BLAST is a protein matching application [2]. The goal is to match an amino acid sequence sample against a large database of proteins. Each processor keeps a portion of the database and tries to match the sample against it. Finally, P.Mem gathers the results.

Mpeg performs MPEG-2 motion estimation. The reference image and the working image are distributed across the processors. Each 8x8 block in the working image is compared against the reference image.

FIC is a fractal image compression application that encodes an image using a scheme with a quad tree partition [6]. Each processor has a portion of the image and some calculated characteristics, and performs a local transformation to its portion of the image. The application may have significant load imbalance.